```
#include <math.h>
#define JMAX 40                    Maximum allowed number of bisections.

float rtbis(float (*func)(float), float x1, float x2, float xacc)
```
Using bisection, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `rtbis`, will be refined until its accuracy is $\pm$`xacc`.
```
{
    void nrerror(char error_text[]);
    int j;
    float dx,f,fmid,xmid,rtb;

    f=(*func)(x1);
    fmid=(*func)(x2);
    if (f*fmid >= 0.0) nrerror("Root must be bracketed for bisection in rtbis");
    rtb = f < 0.0 ? (dx=x2-x1,x1) : (dx=x1-x2,x2);      Orient the search so that f>0
    for (j=1;j<=JMAX;j++) {                                 lies at x+dx.
        fmid=(*func)(xmid=rtb+(dx *= 0.5));             Bisection loop.
        if (fmid <= 0.0) rtb=xmid;
        if (fabs(dx) < xacc || fmid == 0.0) return rtb;
    }
    nrerror("Too many bisections in rtbis");
    return 0.0;                                         Never get here.
}
```

9.2 Secant Method, False Position Method, and Ridders' Method

For functions that are smooth near a root, the methods known respectively as *false position* (or *regula falsi*) and *secant method* generally converge faster than bisection. In both of these methods the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis. After each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root.

The *only* difference between the methods is that secant retains the most recent of the prior estimates (Figure 9.2.1; this requires an arbitrary choice on the first iteration), while false position retains that prior estimate for which the function value has opposite sign from the function value at the current best estimate of the root, so that the two points continue to bracket the root (Figure 9.2.2). Mathematically, the secant method converges more rapidly near a root of a sufficiently continuous function. Its order of convergence can be shown to be the "golden ratio" $1.618\ldots$, so that

$$\lim_{k\to\infty} |\epsilon_{k+1}| \approx \text{const} \times |\epsilon_k|^{1.618} \tag{9.2.1}$$

The secant method has, however, the disadvantage that the root does not necessarily remain bracketed. For functions that are *not* sufficiently continuous, the algorithm can therefore not be guaranteed to converge: Local behavior might send it off towards infinity.

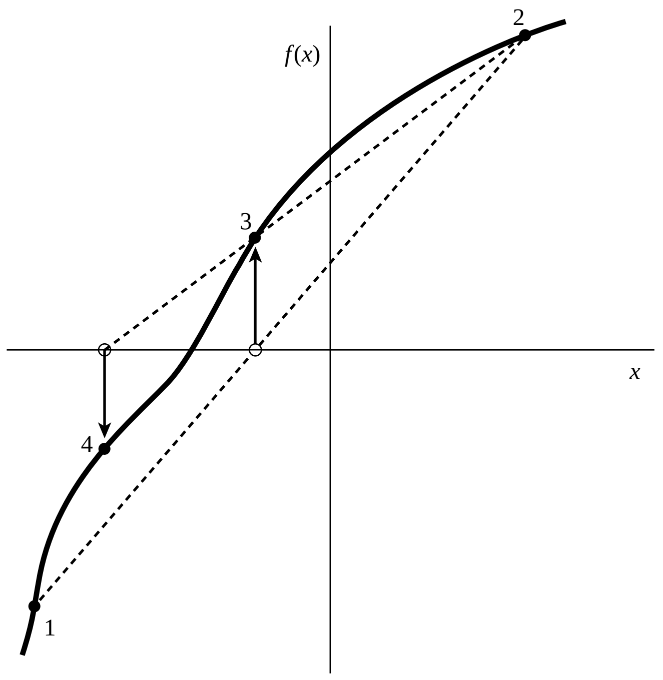

Figure 9.2.1. Secant method. Extrapolation or interpolation lines (dashed) are drawn through the two most recently evaluated points, whether or not they bracket the function. The points are numbered in the order that they are used.

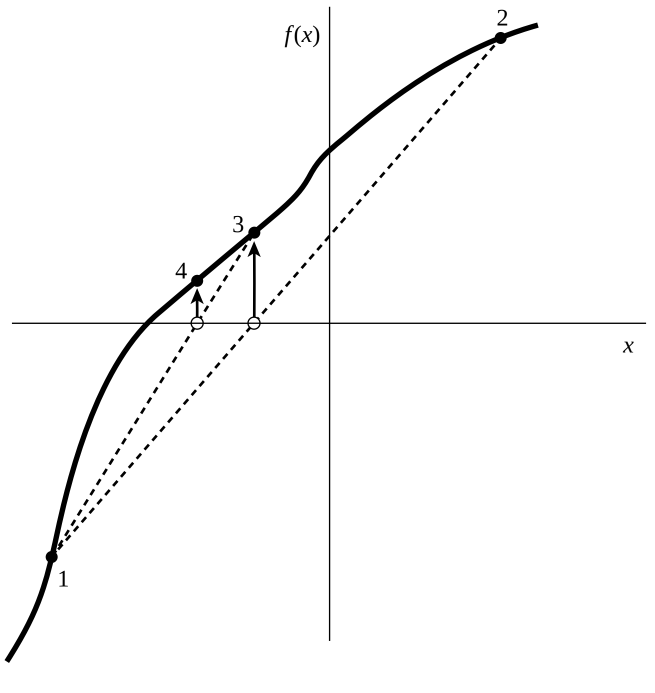

Figure 9.2.2. False position method. Interpolation lines (dashed) are drawn through the most recent points *that bracket the root.* In this example, point 1 thus remains "active" for many steps. False position converges less rapidly than the secant method, but it is more certain.

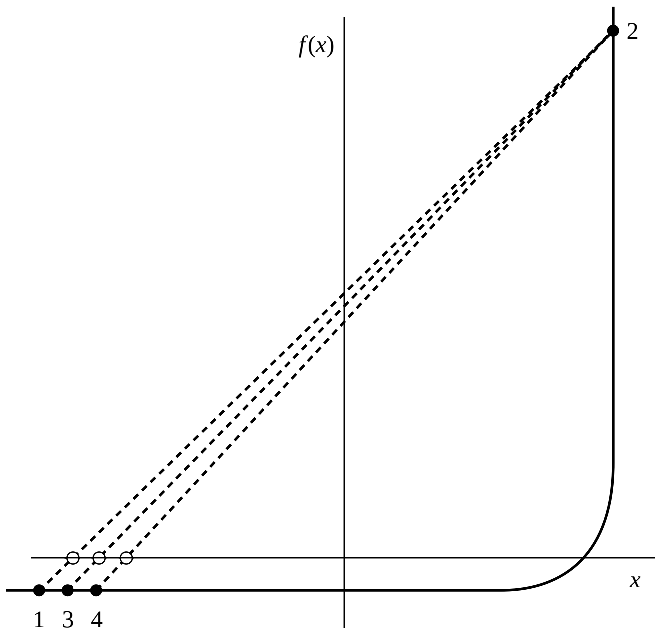

Figure 9.2.3. Example where both the secant and false position methods will take many iterations to arrive at the true root. This function would be difficult for many other root-finding methods.

False position, since it sometimes keeps an older rather than newer function evaluation, has a lower order of convergence. Since the newer function value will *sometimes* be kept, the method is often superlinear, but estimation of its exact order is not so easy.

Here are sample implementations of these two related methods. While these methods are standard textbook fare, *Ridders' method*, described below, or *Brent's method*, in the next section, are almost always better choices. Figure 9.2.3 shows the behavior of secant and false-position methods in a difficult situation.

```
#include <math.h>
#define MAXIT 30                        Set to the maximum allowed number of iterations.

float rtflsp(float (*func)(float), float x1, float x2, float xacc)
```
Using the false position method, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `rtflsp`, is refined until its accuracy is $\pm$`xacc`.
```
{
    void nrerror(char error_text[]);
    int j;
    float fl,fh,xl,xh,swap,dx,del,f,rtf;

    fl=(*func)(x1);
    fh=(*func)(x2);                     Be sure the interval brackets a root.
    if (fl*fh > 0.0) nrerror("Root must be bracketed in rtflsp");
    if (fl < 0.0) {                     Identify the limits so that xl corresponds to the low
        xl=x1;                              side.
        xh=x2;
    } else {
        xl=x2;
        xh=x1;
        swap=fl;
        fl=fh;
        fh=swap;
    }
    dx=xh-xl;
    for (j=1;j<=MAXIT;j++) {            False position loop.
        rtf=xl+dx*fl/(fl-fh);           Increment with respect to latest value.
        f=(*func)(rtf);
        if (f < 0.0) {                  Replace appropriate limit.
            del=xl-rtf;
            xl=rtf;
            fl=f;
        } else {
            del=xh-rtf;
            xh=rtf;
            fh=f;
        }
        dx=xh-xl;
        if (fabs(del) < xacc || f == 0.0) return rtf;       Convergence.
    }
    nrerror("Maximum number of iterations exceeded in rtflsp");
    return 0.0;                         Never get here.
}
```

```
#include <math.h>
#define MAXIT 30                        Maximum allowed number of iterations.

float rtsec(float (*func)(float), float x1, float x2, float xacc)
```
Using the secant method, find the root of a function `func` thought to lie between `x1` and `x2`. The root, returned as `rtsec`, is refined until its accuracy is ±`xacc`.
```
{
    void nrerror(char error_text[]);
    int j;
    float fl,f,dx,swap,xl,rts;

    fl=(*func)(x1);
    f=(*func)(x2);
    if (fabs(fl) < fabs(f)) {           Pick the bound with the smaller function value as
        rts=x1;                             the most recent guess.
        xl=x2;
        swap=fl;
        fl=f;
        f=swap;
    } else {
        xl=x1;
        rts=x2;
    }
    for (j=1;j<=MAXIT;j++) {            Secant loop.
        dx=(xl-rts)*f/(f-fl);           Increment with respect to latest value.
        xl=rts;
        fl=f;
        rts += dx;
        f=(*func)(rts);
        if (fabs(dx) < xacc || f == 0.0) return rts;     Convergence.
    }
    nrerror("Maximum number of iterations exceeded in rtsec");
    return 0.0;                         Never get here.
}
```

Ridders' Method

A powerful variant on false position is due to Ridders [1]. When a root is bracketed between x_1 and x_2, Ridders' method first evaluates the function at the midpoint $x_3 = (x_1 + x_2)/2$. It then factors out that unique exponential function which turns the residual function into a straight line. Specifically, it solves for a factor e^Q that gives

$$f(x_1) - 2f(x_3)e^Q + f(x_2)e^{2Q} = 0 \tag{9.2.2}$$

This is a quadratic equation in e^Q, which can be solved to give

$$e^Q = \frac{f(x_3) + \text{sign}[f(x_2)]\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}{f(x_2)} \tag{9.2.3}$$

Now the false position method is applied, not to the values $f(x_1), f(x_3), f(x_2)$, but to the values $f(x_1), f(x_3)e^Q, f(x_2)e^{2Q}$, yielding a new guess for the root, x_4. The overall updating formula (incorporating the solution 9.2.3) is

$$x_4 = x_3 + (x_3 - x_1)\frac{\text{sign}[f(x_1) - f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}} \tag{9.2.4}$$

Equation (9.2.4) has some very nice properties. First, x_4 is guaranteed to lie in the interval (x_1, x_2), so the method never jumps out of its brackets. Second, the convergence of successive applications of equation (9.2.4) is *quadratic*, that is, $m = 2$ in equation (9.1.4). Since each application of (9.2.4) requires two function evaluations, the actual order of the method is $\sqrt{2}$, not 2; but this is still quite respectably superlinear: the number of significant digits in the answer approximately *doubles* with each two function evaluations. Third, taking out the function's "bend" via exponential (that is, ratio) factors, rather than via a polynomial technique (e.g., fitting a parabola), turns out to give an extraordinarily robust algorithm. In both reliability and speed, Ridders' method is generally competitive with the more highly developed and better established (but more complicated) method of Van Wijngaarden, Dekker, and Brent, which we next discuss.

```
#include <math.h>
#include "nrutil.h"
#define MAXIT 60
#define UNUSED (-1.11e30)

float zriddr(float (*func)(float), float x1, float x2, float xacc)
```
Using Ridders' method, return the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `zriddr`, will be refined to an approximate accuracy `xacc`.
```
{
    int j;
    float ans,fh,fl,fm,fnew,s,xh,xl,xm,xnew;

    fl=(*func)(x1);
    fh=(*func)(x2);
    if ((fl > 0.0 && fh < 0.0) || (fl < 0.0 && fh > 0.0)) {
        xl=x1;
        xh=x2;
        ans=UNUSED;                        Any highly unlikely value, to simplify logic
                                              below.
        for (j=1;j<=MAXIT;j++) {
            xm=0.5*(xl+xh);
            fm=(*func)(xm);                  First of two function evaluations per it-
            s=sqrt(fm*fm-fl*fh);                eration.
            if (s == 0.0) return ans;
            xnew=xm+(xm-xl)*((fl >= fh ? 1.0 : -1.0)*fm/s);     Updating formula.
            if (fabs(xnew-ans) <= xacc) return ans;
            ans=xnew;
            fnew=(*func)(ans);               Second of two function evaluations per
            if (fnew == 0.0) return ans;        iteration.
            if (SIGN(fm,fnew) != fm) {       Bookkeeping to keep the root bracketed
                xl=xm;                          on next iteration.
                fl=fm;
                xh=ans;
                fh=fnew;
            } else if (SIGN(fl,fnew) != fl) {
                xh=ans;
                fh=fnew;
            } else if (SIGN(fh,fnew) != fh) {
                xl=ans;
                fl=fnew;
            } else nrerror("never get here.");
            if (fabs(xh-xl) <= xacc) return ans;
        }
        nrerror("zriddr exceed maximum iterations");
    }
    else {
        if (fl == 0.0) return x1;
        if (fh == 0.0) return x2;
        nrerror("root must be bracketed in zriddr.");
    }
    return 0.0;                              Never get here.
}
```

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.3.

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press), Chapter 12.

Ridders, C.J.F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979–980. [1]

9.3 Van Wijngaarden–Dekker–Brent Method

While secant and false position formally converge faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly. These can be choppy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while secant and false position can sometimes spend many cycles slowly pulling distant bounds closer to a root. Ridders' method does a much better job, but it too can sometimes be fooled. Is there a way to combine superlinear convergence with the sureness of bisection?